Exp No.3
24 2021

Travelling Saleman Problem using Breast First Search

Srinidhi.V
18CI02

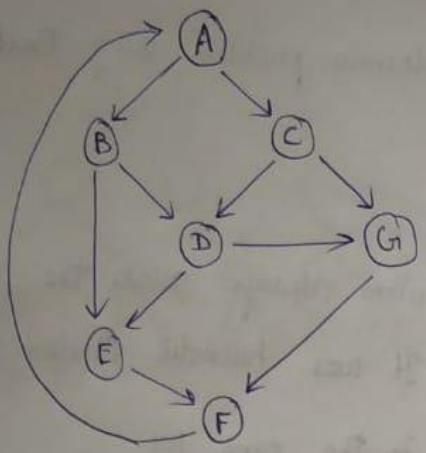
Aim:

To implement travelling saleman problem using Best First Search algorithm.

Algorithm:

* Best first search algorithm always selects the path which appears best at that moment. It uses heuristic function f(n)=h(n)
* Place the starting node in the open list.
* If the open list is empty, stop and return.
* Remove the node n from the open list which has the lowest h(n) and place in any closed list.
* Expand the node n and generate the successors of node n.
* Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate. The search else go to each successor node and check if f(n) and check if the node is not in both the list, then add it to open list.
* Return and check if the list is empty, then stop and return.

Sample Input And Output:



Node	h(n)
A	2
B	5
C	4
D	3
E	4
F	2
G	6

Start Node → A

Best first search algorithm chooses the node with lowest heuristic value.

Node A → Node C [heuristic value = 4]

Node C → Node D [heuristic value = 3]

Node D → Node E [heuristic value = 4]

Node F → Node A [∴ full tour completed]

Path followed by the graph:

A → C → D → E → F → A

Total cost : 15

Program:

```
import heapq as hq

def makeGraph(src, h_val, dest):
    global adj, heur
adj[src] = dest[:]
    heur[src] = h_val

def best_first_search_tsp(src, adj, heur):
pq, path, expanded_nodes = [], [], set()
hq.heapify(pq)
    hq.heappush(pq, (heur[src], src))

    while(len(pq) != 0):
curr_pair = hq.heappop(pq)
curr_heur = curr_pair[0]
curr_node = curr_pair[1]

        if(curr_node == src and src in expanded_nodes): # full tour complete
path.append((curr_node, '$'))
            break         elif(curr_node not in
expanded_nodes):
expanded_nodes.add(curr_node)
path.append((curr_node, curr_heur))
else:
            continue

        for dest in adj[curr_node]:
hq.heappush(pq, (heur[dest], dest))

    return path

def displayResult(path, adj, heur, start):
    print("\nAdjacency List : ", adj)
print("\nHeuristics : ", heur)
print("\nStarting node = ", start)
print("\nPath followed : ")
total_cost = 0     for node, h_val
in path:         if(h_val == "$"):
print(node)         else:
            total_cost += h_val
            print(f"{node} ({h_val}) -> ", end=" ")

    print("Total Cost = ", total_cost)


if __name__ == "__main__":
    adj, heur = dict(), dict()
    makeGraph(src = 'A', h_val = 2, dest = ['D', 'C'])
makeGraph(src = 'B', h_val = 5, dest = ['E'])
```

```
makeGraph(src = 'C', h_val = 4, dest = ['B', 'E'])
makeGraph(src = 'D', h_val = 3, dest = ['C'])
makeGraph(src = 'E', h_val = 6, dest = ['D', 'A'])     start
= 'A'
    path = best_first_search_tsp(start, adj, heur)
displayResult(path, adj, heur, start)
```

Output:

```
Adjacency List :  {'A': ['D', 'C'], 'B': ['E'], 'C': ['B', 'E'], 'D': ['C'], 'E': ['D', 'A']}

Heuristics :  {'A': 2, 'B': 5, 'C': 4, 'D': 3, 'E': 6}

Starting node =  A

Path followed :
A (2) ->  D (3) ->  C (4) ->  B (5) ->  E (6) ->  A
Total Cost =  20
```

Result:

Thus, travelling salesman problem is implemented using best first search.